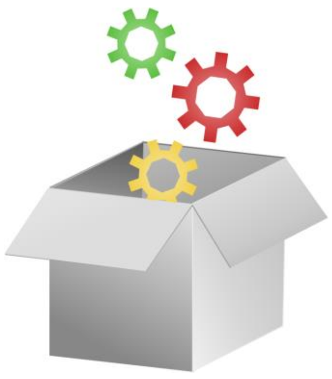
Method SCRIPT Examples



Last revision: March ? 2019

Commented [P1]: Tbu

www.palmsens.com

1.1 Contents:

The examples in the "/Method SCRIPT Examples" folder demonstrate basic communication with the EmStat Pico like how to connect to the device, send method scripts to the device, run measurements on the device, read the response from the device, using simple plot objects to plot the data and close a connection to the device.

1.2 Examples:

1.2.1 Example 1: Basic Console Example (MSConsoleExample)

This example demonstrates the easiest and most minimalistic way of implementing USB serial communication to

- Write the input parameters for a measurement (LSV technique) read from a script file
- Read the measurement response from the device.
- Parse the response and display the results on the console.

This does not include error handling, method validation etc.

1.2.2 Example 2: Plot Example (MSPlotExample)

This example demonstrates how to implement the plot object (using the OxyPlot library for windows forms) and USB communication as well as how to receive and plot curve-data during measurements.

1.2.3 Example 3: EIS Console Example (MSEISExample)

This console example demonstrates the send, receive and parsing data for a simple EIS measurement.

1.2.4 Example 4: EIS Plot Example (MSEISPlotExample)

This example demonstrates how to implement the plot object (using the OxyPlot library for windows forms) and USB communication as well as how to receive and plot curve-data of a simple EIS measurement.

1.3 Communications

The example projects are built using Visual Studio 17.

1.3.1 Connecting to the device

The following example shows how to get a list of all available serial com ports using the System.IO.Ports library and how to identify the serial port connected to the EmStatPico. The command "t\n" is sent to the port to receive the version string from the device and If the response contains the string "esp", it is identified as EmStat Pico. The read time out for the port is then set to 7000ms. The code below will identify and return the serial port connected to the EmStat Pico.

```
private static SerialPort OpenSerialPort()
{
    SerialPort serialPort = null;
    string[] ports = SerialPort.GetPortNames();
    for (int i = 0; i < ports.Length; i++)
    {
        serialPort = GetSerialPort(ports[i]);
        try
        {
            serialPort.Open();                  // Open serial port
            if (serialPort.IsOpen)
            {
                serialPort.Write("t\n");
                string response = serialPort.ReadLine();
                // Identify the port connected to EmStatPico
                if (response.Contains("esp"))
                {
                    serialPort.ReadTimeout = 7000;
                    return serialPort;
                }
            }
        }
        catch (Exception exception)
        {
             serialPort.Close();
        }
    }
    return serialPort;
}
```

To prevent your program from crashing it is recommended to use a try catch sequence when connecting to a port, this way the port will be closed again when an exception occurs.

1.3.2 Sending a script file

The input parameters for a measurement can be read from a script file stored in the path "ScriptFilePath". In this example the script file is stored in the same location as the location of the application executable.

```
// Name of the script file
static string ScriptFileName = "meas_swv_test.txt";//"LSV_on_1KOhm.txt";
static string AppLocation = Assembly.GetExecutingAssembly().Location;
static string FilePath = System.IO.Path.GetDirectoryName(AppLocation) +
\\scripts";
// Path of the script file
static string ScriptFilePath = Path.Combine(FilePath, ScriptFileName);
```

The contents of the script file can be then be read line by line using StreamReader and written on the device.

```
    string line = "";
    using (StreamReader stream = new StreamReader(ScriptFilePath))
    {
        while (!stream.EndOfStream)
        {
            line = stream.ReadLine();  // Read a line from the script file
            line += "\n";    // Append a new line character to the line read
            SerialPortEsP.Write(line); // Send the read line to EmStat Pico
        }
        Console.WriteLine("Measurement started.");
    }
```

1.3.3 Receiving response

Once the script file is sent to the device, the measurement response can be read continuously from the device. In the code below, the response is read line by line ('\n' is considered to be the end of line character) and each line is parsed to retrieve the actual data values.

```
private static string ReadResponseLine()
{
    string readLine = "";
    int readChar;
    while (true)
    {
       // Read a character from the serial port input buffer
        readChar = SerialPortEsP.ReadChar();
      // Possibility of time out exception if the operation doesn't
complete within the read time out
        if (readChar > 0)
        {
           // Append the read character to readLine to form a response line
            readLine += (char)readChar;
            if ((char)readChar == '\n')
            {
               // Return the readLine when a new line character is
            encountered
                return readLine;
            }
        }
    }
}
```

1.3.4 Parsing the response

Each line of response returned by the method `ReadResponseLine()`, can be added to a string to form the raw data if needed. The response line can be further parsed if it is identified to be a data package.

```
while (true)
{
    readLine = ReadResponseLine();               // Read a line from the
response
    RawData += readLine;                         // Add the response to raw
data
    if (readLine == "\n")
        break;
    else if (readLine.Contains("P"))
    {
//Increment the number of data points if the read line contains the header
char 'P'
        NDataPointsReceived++;
        ParsePackageLine(readLine);             // Parse the line read
    }
}
```

Here's a sample response (raw data) from a Linear sweep voltammetric measurement.

```
eM0000\n
Pda7F85F3Fu;ba48D503Dp,10,288\n
Pda7F9234Bu;ba4E2C324p,10,288\n
Pda806EC24u;baAE16C6Dp,10,288\n
```

Pda807B031u;baB360495p,10,288\n
*\n
\n
While parsing the response, various identifiers are used to identify the type of response packages. For example, In the above sample response package,

1. 'e' marks the beginning of a response.
2. 'M' marks the beginning of a measurement loop.
3. 'P' marks the beginning of a row of data package.
4. "*\n" marks the end of measurement.
5. "\n" marks the end of response.

The following information can be found in the data packages received from the device.

- Potential (set cell potential in V)
- Current (measured current in A)
- In case of Impedance spectroscopy measurements, the following data values can be obtained from the response
- Frequency (set frequency in Hz)
- Real part of complex Impedance (measured impedance Ohm)
- Imaginary part of complex Impedance (measured impedance in Ohm)

The following meta data values if present can also be obtained from the data packages.

- CurrentStatus (OK, underload, overload, overload warning)
- CurrentRange (the current range in use at the moment)
- Noise (Noise)

1.3.4.1 Parsing the parameter values

Each row of data package begins with the header 'P' which is removed first.

```
    int startingIndex = packageLine.IndexOf('P');
    string responsePackageLine = packageLine.Remove(startingIndex, 1);
```

The rest of the package is contains the parameters which are then separated by the delimiter character ';'

```
    parameters = responsePackageLine.Split(';');
```

Each of the parameters are then parsed.
The initial two characters of every parameter identifies the parameter.

```
    paramIdentifier = parameter.Substring(startingIndex, 2);
```

For example, in the sample package seen above, the parameter identifiers are
'da7F85F3Fu' - 'da' Potential reading and
'ba48D503Dp,10,288' – 'ba' current reading.

The parameter values hold the next 8 characters.

```
const int PACKAGE_PARAM_VALUE_LENGTH = 8;

paramValue = responsePackageLine.Substring(startingIndex + 2,
PACKAGE_PARAM_VALUE_LENGTH);
```

The parameter value for current reading (8 characters)from the above sample package is '48D503Dp'.
This value is further parsed to retrieve the actual parameter value with the respective unit prefix.

```
double paramValueWithPrefix = ParseParamValues(paramValue);
```

The SI unit prefix from the package can be obtained from the parameter value at position 8

```
    char strUnitPrefix = paramValueString[7];
```

In the above sample package, the unit prefix for current data is ‘p’ which is 1e-12 A.

The code below parses the actual parameter value excluding the unit prefix (7 characters) and appends the respective prefixes.

```
string strvalue = paramValueString.Remove(7);
```

The value is first converted from hex to int

```
int value = Convert.ToInt32(strvalue, 16);
```

Then value is then adjusted with the Offset value to receive only positive values.

```
const int OFFSET_VALUE = 0x8000000;
double paramValue = value - OFFSET_VALUE;
```

The value of the parameter is returned after appending the SI unit prefix

```
return (paramValue * SI_Prefix_Factor[strUnitPrefix.ToString()]);
```

The SI unit prefixes are as follows.

```
readonly static Dictionary<string, double> SI_Prefix_Factor = new
Dictionary<string, double>
                  { { "a", 1e-18 },
                    { "f", 1e-15 },
                    { "p", 1e-12 },
                    { "n", 1e-9 },
                    { "u", 1e-6 },
                    { "m", 1e-3 },
                    { " ", 1 },
                    { "K", 1e3 },
                    { "M", 1e6 },
                    { "G", 1e9 },
                    { "T", 1e12 },
                    { "P", 1e15 },
                    { "E", 1e18 }};
```

The parameter values can be added to the corresponding arrays based on the parameter identifiers.
The potential readings are identified by the string “da”
The current readings are identified by the string “ba”

```
switch (paramIdentifier)
        {
            case "da":                                // Potential reading
      // If potential reading add the value to the VoltageReadings array
                VoltageReadings.Add(paramValueWithPrefix);
                break;
            case "ba":                                // Current reading
      // If current reading add the value to the CurrentReadings array
                CurrentReadings.Add(paramValueWithPrefix);
                break;
        }
```

In case of Impedance sprctroscopy measurement, the following identifiers are used.
The frequency readings are identified by the string “dc”
The real impedance readings are identified by the string “cc”
The imaginary impedance readings are identified by the string “cd”

```
switch (paramIdentifier)
{
    case "dc":                                   //Frequency reading
    //If frequency reading add the value to the FrequencyReadings list
        FrequencyValues.Add(paramValueWithPrefix);
        break;
    case "cc":                               //Real Impedance reading
    //If Z(Real) reading add the value to RealImpedanceReadings list
        RealImpedanceValues.Add(paramValueWithPrefix);
        break;
    case "cd":                           //Imaginary Impedance reading
    //If Z(Img) reading add the value to ImgImpedanceReadings list
        ImgImpedanceValues.Add(paramValueWithPrefix);
        break;
}
```

After obtaining the parameter identifier and the parameter values from the package, the meta data values if present can be parsed. Meta data values if present are separated by the demiliter character ‘,’

```
ParseMetaDataValues(parameter.Substring(10));
```

1.3.4.2 Parsing the meta data values

The meta data values are separated based on the delimiter ‘,’ and each of the values is further parsed to get the actual value. The first character of each meta data value identifies the type of meta data.

‘1’ – status
‘2’ – Current range index
‘4’ - Noise

The status is 1 character hex bit mask. It is converted to long int.

The status can be obtained as shown in the code below.

```
string status = "";
long statusBits = (Convert.ToInt32(metaDatastatus[1].ToString(), 16));
```

0 indicates OK

```
if ((statusBits & 0x0) == (long) ReadingStatus.OK)
    status = "OK";
```

1 indicates overload

```
if ((statusBits & 0x2) == (long) ReadingStatus.Overload)
    status = "Overload";
```

4 indicates underload

```
if ((statusBits & 0x4) == (long) ReadingStatus.Underload)
    status = "Underload";
```

and 8 indicates overload warning (80% of maximum).

```
if ((statusBits & 0x8) == (long) ReadingStatus.Overload_Warning)
    status = "Overload warning";
```

For example, in the above sample, the available meta data values for current data are, 10,288. The first meta data value is 10.

1 – meta data status – 0 indicates OK.

The meta data type current range is 2 characters long hex value. If the first bit high (0x80) it indicates a high speed mode current range.

The possible current ranges supported by the EmStat Pico are as in the code below.

```
public enum CurrentRanges
{
    cr100nA = 0,
    cr2uA = 1,
    cr4uA = 2,
    cr8uA = 3,
    cr16uA = 4,
    cr32uA = 5,
    cr63uA = 6,
    cr125uA = 7,
    cr250uA = 8,
    cr500uA = 9,
    cr1mA = 10,
    cr5mA = 11,
    //High speed mode current ranges
    hscr100nA = 128,
    hscr1uA = 129,
    hscr6uA = 130,
    hscr13uA = 131,
    hscr25uA = 132,
    hscr50uA = 133,
    hscr100uA = 134,
    hscr200uA = 135,
    hscr1mA = 136,
    hscr5mA = 137,
}
```

The code below can be used to get current range from the package.

```
string currentRangeStr = "";
byte crByte;
if(byte.TryParse(metaDataCR.Substring(1,2), NumberStyles.AllowHexSpecifier,
CultureInfo.InvariantCulture, out crByte))
{
switch (crByte)
{
    case (byte)CurrentRanges.cr100nA:
        currentRangeStr = "100nA";
        break;
    case (byte)CurrentRanges.cr2uA:
        currentRangeStr = "2uA";
        break;
    case (byte)CurrentRanges.cr4uA:
        currentRangeStr = "4uA";
        break;
    case (byte)CurrentRanges.cr8uA:
        currentRangeStr = "8uA";
        break;
    case (byte)CurrentRanges.cr16uA:
        currentRangeStr = "16uA";
        break;
    case (byte)CurrentRanges.cr32uA:
        currentRangeStr = "32uA";
        break;
    case (byte)CurrentRanges.cr63uA:
        currentRangeStr = "63uA";
        break;
    case (byte)CurrentRanges.cr125uA:
        currentRangeStr = "125uA";
        break;
    case (byte)CurrentRanges.cr250uA:
        currentRangeStr = "250uA";
        break;
    case (byte)CurrentRanges.cr500uA:
        currentRangeStr = "500uA";
        break;
    case (byte)CurrentRanges.cr1mA:
        currentRangeStr = "1mA";
        break;
    case (byte)CurrentRanges.cr5mA:
        currentRangeStr = "15mA";
        break;
    case (byte)CurrentRanges.hscr100nA:
        currentRangeStr = "100nA (High speed)";
        break;
    case (byte)CurrentRanges.hscr1uA:
        currentRangeStr = "1uA (High speed)";
        break;
    case (byte)CurrentRanges.hscr6uA:
        currentRangeStr = "6uA (High speed)";
        break;
    case (byte)CurrentRanges.hscr13uA:
        currentRangeStr = "13uA (High speed)";
        break;
    case (byte)CurrentRanges.hscr25uA:
        currentRangeStr = "25uA (High speed)";
        break;
    case (byte)CurrentRanges.hscr50uA:
        currentRangeStr = "50uA (High speed)";
        break;
    case (byte)CurrentRanges.hscr100uA:
        currentRangeStr = "100uA (High speed)";
        break;
    case (byte)CurrentRanges.hscr200uA:
        currentRangeStr = "200uA (High speed)";
        break;
    case (byte)CurrentRanges.hscr1mA:
        currentRangeStr = "1mA (High speed)";
        break;
    case (byte)CurrentRanges.hscr5mA:
        currentRangeStr = "5mA (High speed)";
        break;
}
```

For example, in the above sample, the second meta data available is 288.

2 – indicates the type – current range
88 – indicates the hex value for current range index – 1mA. The first bit 8 implies that it is high speed mode current range.

Sample outputs:

The following are some sample data packages and their correponding outputs.

(Please note that the spaces in the data packages are added for readability).
A row of data package from LSV measurement response

P da7F85F3Fu ; ba4BA99F0p , 10 , 288

```
Output:     E (V) = -4.999E-01
            i (A) = -4.999E-01
            Status : OK
            CR : 1mA (High speed)
```

A row of data package fom EIS measurement response

P dc8030D40 ; cc8088C7Cm , 12 , 289 ; cdA9279D1u ,12 , 289

```
Output:     Frequency (Hz) : 200000.00
            Z' (Ohm) : 2.000E+05    Status : Overload       CR : 5mA
            Z'' (Ohm) : 2.000E+05   Status : Overload       CR : 5mA
```

1.3.5 Plotting the response

The OxyPlot library from NuGet packages is used in this example for showing a simple plot of the measurement response parameters.

```
using OxyPlot;
using OxyPlot.Axes;
using OxyPlot.Series;
```

A new instance of PlotModel is created and set as the model to the plot object created in the UI.

```
private PlotModel plotModel = new PlotModel();
samplePlotView.Model = plotModel;
```

A new instance of LineSeries is also created and added to the plot model.

```
private LineSeries plotData;
plotData = new LineSeries()
            {
                Color = OxyColors.Green,
                MarkerType = MarkerType.Circle,
                MarkerSize = 6,
                MarkerStroke = OxyColors.White,
                MarkerFill = OxyColors.Green,
                MarkerStrokeThickness = 1.5,
            };
plotModel.Series.Add(plotData);
```

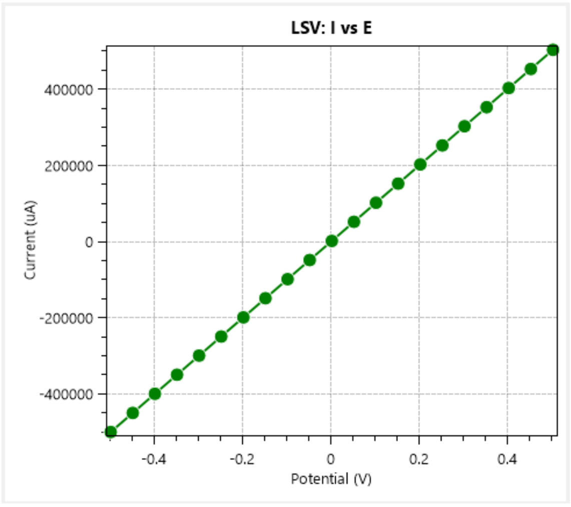
The axes for the plot model can be set as shown in the code below.

```
private void SetAxes()
{
    var xAxis = new LinearAxis()
    {
        Position = OxyPlot.Axes.AxisPosition.Bottom,
        MajorGridlineStyle = LineStyle.Dash,
        Title = "Potential (V)"
    };
    var yAxis = new OxyPlot.Axes.LinearAxis()
    {
        Position = OxyPlot.Axes.AxisPosition.Left,
        MajorGridlineStyle = LineStyle.Dash,
        Title = "Current (uA)"
    };
    //Set the x-axis and y-axis for the plot model
    plotModel.Axes.Add(xAxis);
    plotModel.Axes.Add(yAxis);
}
```

As and when a new line of response is read and parameter values are added to their corresponding arrays, the plot is updated.

```
// Add the last added measurement values as new data points and update the
plot
plotData.Points.Add(new DataPoint(VoltageReadings.Last(),
CurrentReadings.Last()));
plotModel.InvalidatePlot(true);
```

Here's a sample plot with the response for a Linear sweep measurement on EmStat Pico.



In case of an Impedance spectroscopy measurement, two separate plot objects and plot models are to be created. One for Nyquist plot and the other for Bode plot.

The plot type is set to 'Cartesian' type in case of Nyquist plot model and 'XY' type in case of Bode plot model.

```
plotModel.PlotType = plotModel.Equals(NyquistPlotModel) ?
PlotType.Cartesian : PlotType.XY;
```

The plot axes for both the plot models are set as below.

Nyquist plot axes are set to linear axes.

```
    var xAxisNyquistPlot = GetLinearAxis("Z (Ohm)", AxisPosition.Bottom);
    var yAxisNyquistPlot = GetLinearAxis("-Z'' (Ohm)", AxisPosition.Left);
```

```
    //Add the axes to the Nyquist plot model
    NyquistPlotModel.Axes.Add(xAxisNyquistPlot);
    NyquistPlotModel.Axes.Add(yAxisNyquistPlot);
```

Bode plot axes are set to logarithmic axes.

```
    var xAxisBodePlot = GetLogAxis("Frequency", "Frequency (HZ)",
AxisPosition.Bottom, OxyColors.Black);
    var yAxisBodePlot = GetLogAxis("Z", "Z (Ohm)", AxisPosition.Left,
OxyColors.Blue);
    var yAxisSecondaryBodePlot = GetLogAxis("Phase", "-Phase (deg)",
AxisPosition.Right, OxyColors.Red);
```

```
    //Add the axes to the Bode plot model
    BodePlotModel.Axes.Add(xAxisBodePlot);
    BodePlotModel.Axes.Add(yAxisBodePlot);
    BodePlotModel.Axes.Add(yAxisSecondaryBodePlot);
```

The magnitude of and phase of impedance is required for the Bode plot. This can be obtained by calculating the complex impedance value the real and imaginary parts of the impedance retrieved from the package as in the code below.

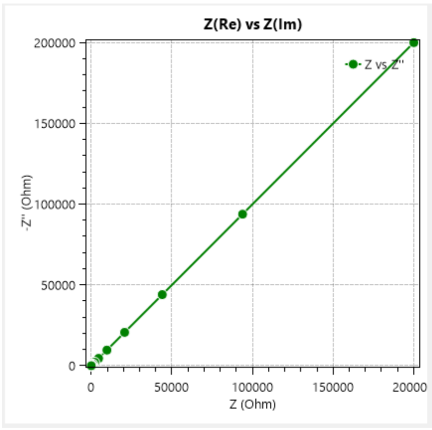
```
Complex ZComplex = new Complex(RealImpedanceValues.Last(),
ImgImpedanceValues.Last());
ComplexImpedanceValues.Add(ZComplex);
ImpedanceMagnitudeValues.Add(ComplexImpedanceValues.Last().Magnitude);
PhaseValues.Add(ComplexImpedanceValues.Last().Phase * 180 / Math.PI);
```

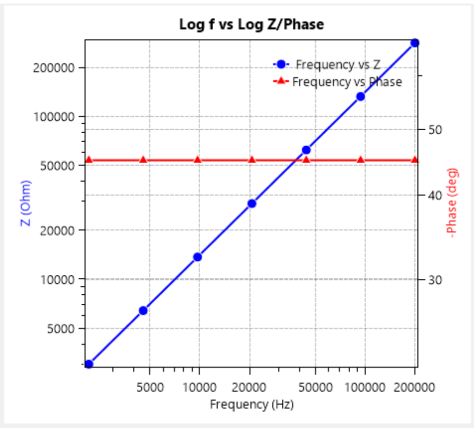
`The plots are then updated with the data values.`

```
NyquistPlotData.Points.Add(new DataPoint(RealImpedanceValues.Last(),
ImgImpedanceValues.Last()));    // Add the last added measurement values as
new data points and update the plot
NyquistPlotModel.InvalidatePlot(true);
NyquistPlotModel.ResetAllAxes();
```

```
BodePlotDataMagnitude.Points.Add(new DataPoint(FrequencyValues.Last(),
ImpedanceMagnitudeValues.Last()));
BodePlotDataPhase.Points.Add(new DataPoint(FrequencyValues.Last(),
PhaseValues.Last()));BodePlotModel.InvalidatePlot(true);
BodePlotModel.ResetAllAxes();
```

Here's are the Nyquist and Bode plots with the response for an Impedance sprctroscopy measurement on EmStat Pico.


Z(Re) vs Z(Im)
Z vs Z''
-Z'' (Ohm)
Z (Ohm)
0
50000
100000
150000
200000
20000


Log f vs Log Z/Phase
Frequency vs Z
Frequency vs Phase
Z (Ohm)
-Phase (deg)
Frequency (Hz)
5000
10000
20000
50000
100000
200000
30
40
50